A TRIDENT SCHOLAR PROJECT REPORT

NO. 440

Machine Learning Based Malware Detection

by

Midshipman 1/C Zane A. Markel, USN

UNITED STATES NAVAL ACADEMY
ANNAPOLIS, MARYLAND

USNA-1531-2

U.S.N.A. --- Trident Scholar project report; no. 440 (2015)

MACHINE LEARNING BASED MALWARE DETECTION

by

Midshipman 1/C Zane A. Markel
United States Naval Academy
Annapolis, Maryland

_____

Certification of Adviser(s) Approval

Permanent Military Professor CDR Michael B. Bilzor, USN
Computer Science Department

_____

_____

Acceptance for the Trident Scholar Committee

Professor Maria J. Schroeder
Associate Director of Midshipman Research

_____

_____

USNA-1531-2

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
05-18-2015		

4. TITLE AND SUBTITLE
Machine Learning Based Malware Detection

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Markel, Zane Alexander

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Naval Academy
Annapolis, MD 21402

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Trident Scholar Report no. 440 (2015)

12. DISTRIBUTION / AVAILABILITY STATEMENT

This document has been approved for public release; its distribution is UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Current antivirus software is effective at detecting well known threats but cannot keep up with the rate at which new malware is authored nor modern antivirus avoidance techniques, such as using polymorphic code. Some studies have investigated augmenting current antivirus techniques with machine learning, which could potentially detect some previously unknown malware. However, previously proposed methods either do not detect malware with satisfactory performance, or they have only been tested on laboratory software databases that cannot suitably be projected into realistic performance.

This work explores several aspects of machine learning based malware detection. First, we propose an approach to learn primarily from program metadata, particularly header data in the 32-bit Windows Portable Executable (PE32) file format. We identify learning methods that learn effectively from this metadata, explore which metadata features can be trivially modified and are not appropriate for malware detection, test it on approximately realistic datasets, and find that it performs favorably compared to Windows API imports, another category of file characteristic that shows promise for machine learning based malware detection. Additionally, we find and explore the drastic performance drop which occurs when using a realistically low proportion of malware in test datasets instead of datasets split evenly between malware and benign software.

15. SUBJECT TERMS
malware, machine learning, class imbalance, ensemble learning, portable executable, Windows API

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		24	**19b. TELEPHONE NUMBER** *(include area code)*

1 Abstract

Current antivirus software is effective at detecting well known threats but cannot keep up with the rate at which new malware is authored nor modern antivirus avoidance techniques, such as using polymorphic code. Some studies have investigated augmenting current antivirus techniques with machine learning, which could potentially detect some previously unknown malware. However, previously proposed methods either do not detect malware with satisfactory performance, or they have only been tested on laboratory software databases that cannot suitably be projected into realistic performance.

This work explores several aspects of machine learning based malware detection. First, we propose an approach to learn primarily from program metadata, particularly header data in the 32-bit Windows Portable Executable (PE32) file format. We identify learning methods that learn effectively from this metadata, explore which metadata features can be trivially modified and are not appropriate for malware detection, test it on approximately realistic datasets, and find that it performs favorably compared to Windows API imports, another category of file characteristic that shows promise for machine learning based malware detection.

Additionally, we find and explore the drastic performance drop which occurs when using a realistically low proportion of malware in test datasets instead of datasets split evenly between malware and benign software. Ensemble learning, which commonly alleviates this problem in other similar machine learning applications, does not appreciably help in this context. Training with datasets that have the same proportion of malware as the test datasets optimizes performance, yet the file characteristics that are informative for malware detection change with the proportion of malware in the training dataset. We conclude that file characteristics must be trained on and tested in approximately realistic settings in order to demonstrate their robustness in operational malware detection, and we propose a test procedure which meets these standards.

Key words: malware, machine learning, class imbalance, ensemble learning, portable executable, Windows API

2 Acknowledgments

The authors would like to thank the U.S. Naval Academy Research Office and Trident Scholar program for their generous support.

Table of Contents

3 Introduction

Modern antivirus software is effective at detecting known threats but can be evaded by specially crafted novel malware such as polymorphic malware, which can reprogram itself to appear and operate differently while ultimately performing the same functions overall. Traditional antivirus programs use signature-based detection, which involves checking potential malware against a database of *hashes*—fingerprints—of the exact files of known malware, yet this technique has severe limitations [1], [2]. Modern antivirus products also employ both static heuristic checks and dynamic analysis, which use manually crafted rules to detect malware based on code structure or behavior. However, a variety of techniques have been published that defeat both heuristic and dynamic analysis, as well [3].

3.1 Machine Learning in Malware Classification

One way to potentially keep up with the antivirus avoidance techniques used in modern malware is to augment existing detection systems with machine learning classifiers. Machine learning classification algorithms facilitate construction of *classifiers*, which automatically learn the characteristics of each class, such as malware and benign files, by learning from example data. This kind of approach, while still developmental in malware detection, offers the promise of increased robustness in the face of newly adapted threats that are slightly modified, but retain some of the characteristics of past malware.

To use machine learning to classify executable files as malicious or benign, one must first build labeled datasets for training. A set of *records*—one for each benign and malicious file—comprises the database. Each record contains a set of *features* and one *label* (also referred to as the *class*). The features of each file are derived from some specific characteristics of the file, such as the size of the file or the frequency of a certain code snippet in the file; the label is a binary value indicating whether or not the file is malicious. Learning algorithms analyze records designated for training to generate a mathematical model that maps the relationship of file features and labels. That model, which is called the *classifier*, is used to predict the class of each record in the *test data*, or the records designated for testing. The classifier cannot read the labels when making predictions; test data labels are only used when the predictions are compared against the true labels in subsequent analysis of performance.

A variety of features have been studied as potentially effective discriminators between malware and benign software. These features can be extracted either through dynamic or static analysis. Static analysis extracts features either from the header, which contains metadata, or the body, which contains the actual code and data, of an executable file. Dynamic analysis involves carefully executing a program in a safe environment and measuring features by recording behavior such as interactions with the operating system or network traffic. While dynamic analysis can sometimes reveal behavior in malware that would be difficult to extract from static analysis [4], dynamic analysis is also more resource intensive than some types of static analysis [5], and can currently be defeated in a variety of ways [3].

3.2 Related Results

A number of researchers have used machine learning to classify malware. Three types of features that have been popular in experiments to date are n-grams over machine code instructions, API call sequences, and PE32 header data.

N-gram analysis involves splitting a program's code into chunks of size "n"—the n-grams. File features are derived from frequencies of certain n-grams. Some studies have achieved modest success with this technique [6], [7]. However, it has also been argued that, both from a theoretic and empirical perspective, n-gram frequencies do not differ enough between malicious and benign software to be useful for practical malware detection [8]. That is, machine learning cannot effectively differentiate malware and benign software by looking at isolated tiny snippets of machine code.

A classifier can also be constructed using features derived from function calls from the operating system API, which are functions a program uses to interact with the operating system. These features can either be static, which involve examining which API functions a file imports into its code, or dynamic, which involve examining sequences of API functions used during execution. Belaoued and Mazouzi show that the difference between Windows API imports in malware and benign software is statistically significant [9]. However, there are methods for invoking OS functionality while bypassing the API [10], and it is not clear whether the discriminative power of API calls or imports alone will be sufficient for an operational malware classifier.

For Windows Portable Executable (PE32) files, the format for executable files on Windows operating systems, header data has been found to discriminate well between malicious and benign executables. PE32 header data contains many fields which describe the structure of the executable file and metadata about the file, such as how large the executable code section is or what year the file was created. (For a full description of PE32 header information, see the PE specification [11].) In 2012, Yonts published statistical comparisons between the header data of clean and malicious PE32 files [12], showing that malicious and benign programs frequently differ in certain header components. Yonts also manually designed "detection rules" that individually discriminated reasonably well between the malware and benign software in his database [12], but did not use the features in aggregate in the way a machine learning classifier would. Other studies have highlighted the potential value of PE32 header data as well, yet no previous research has proposed a classifier to discriminate between benign and malicious software in general [5], [13], [14]. PE32 header data is also an attractive feature source because it can be extracted quickly. Some researchers, including Walenstein et al. and Yan et. al, note that certain PE32 header data could be easily modified by an attacker, rendering such features ineffective in the future [5], [13].

Many studies have built machine learning classifiers that have achieved high performance in lab experiments, but the conditions used do not reflect practical applications. In the real world, detecting malware hiding on a computer system with thousands of benign executable files is more like finding a needle in a haystack. When building classifiers, studies to date have tended to

use a large number of both malware and benign software in the training data[1], even though benign software outnumbers malware by orders of magnitude on most general use computers. However, it is not fair to assume that results obtained from training or testing on balanced proportions of malware and benign software imply similar results when training or testing on more realistic ratios where benign software greatly outnumbers malware. For machine learning in general, it has been widely observed that *class imbalance*, in which one class greatly outnumbers the other in one's database, often diminishes classifier performance.

Guo et al. surveyed class imbalance studies, and identified remedies at various phases of the learning process. They gave several suggestions for improving the preliminary training stages, and pointed out that ensemble learning algorithms are commonly used to address class imbalance. Ensemble methods, which train several classifiers to make aggregate classification predictions, make predictions in a variety of ways, including "voting" by individual classifiers or randomly choosing one classifier to use for the final prediction. The study by Guo et al. specifically highlights techniques called Boosting and Bagging for their success in learning with imbalanced classes [15].

4 Hypotheses and Goals

This research contributes to the study of machine learning based malware detection through several phases. First, it explores the efficacy of using features based on PE32 header data for malware detection. We hypothesize that classifiers trained on PE32 header data will perform at least as well as previously published malware detection classifiers. To show this, a variety of learning algorithms and methods are trained on PE32 header data features in a procedure comparable to the predominant ones in the field. Through these experiments, show that, when using the predominant methods in the literature, PE32 header data appears to be an effective source of features for machine learning based malware detection.

The next phase of this research addresses the class imbalance problem. First, we conduct initial experiments to gauge the performance drop that occurs when a realistically low proportion of test data represents malware versus when the test data is split relatively evenly between malware and benign software. Furthermore, various methods for mitigating the performance drop when malware prevalence in the test data is low are explored on two separate and independent sets of features. We hypothesize that classification performance will drop substantially when classifiers test on data with a realistically low proportion of malware, and we further predict this problem will be resilient to techniques that have helped to mitigate the effects of class imbalance in other machine learning applications. If true, this finding would imply that more research will be needed to address the class imbalance problem for machine learning based malware detection in order for the technique to become useful in practical settings.

[1] For a concise summary of the datasets used in previous malware classification work that involves header based features, see Walenstein et al. [13].

Finally, we question the assumption that features which have been demonstrated to be useful for machine learning based malware detection when the training data does not contain class imbalance are guaranteed to be useful when the training data does have a class imbalance. To our knowledge, no previous work has questioned this assumption. Experiments will be conducted to assess the independence of relative feature utility and the prevalence of malware in the training data. We hypothesize that the assumption will not hold. If so, we will propose novel criteria for demonstrating the robustness of a given feature for machine learning based malware detection.

5 Methodology

5.1 Data

Before our experiments could begin, a database of known clean and malicious PE32 files was needed. The malicious files were randomly selected from a collection we obtained from Open Malware, a group dedicated to safely distributing known malware for research purposes [16]. In total, we scanned 122,799 unique malicious files in constructing our database. To obtain a diverse sample of benign files, we scanned all the PE32 files in the `C:\Windows` and `C:\Program Files` directories from the following clean installations of Microsoft Windows:

- Windows Vista Enterprise
- Windows 7 Professional
- Windows Server 2008 R2 Standard
- Windows 8.1 Professional

Additionally, we scanned the PE32 files from clean installations of a diverse set of 46 known benign Windows applications such as LibreOffice, FileZilla, Chrome, Firefox, QuickTime, Microsoft Office, Python, Java, VLC, etc. [17], [18]. After removing duplicates, our database contained 42,003 benign files, for a collection of 164,802 files total.

We used `pefile` [19], an extension to the Python programming language, to write a program that would measure 38 different features from the PE32 headers, plus 2 features derived from the section entropies of a PE32 file, for any file given to it. These features were chosen for their discriminatory potential as identified by Yonts [12] and because they cannot be trivially modified without affecting file execution. This program was used to build a database containing the values of all 40 features for each of our 122,799 files. We also used `pefile` to extract the Windows API imports of each file and built a separate database with a feature for each of the 2,067 uniquely named Windows API functions indicating whether or not a file imported the API function corresponding to that feature [20]. Each database was structured as a set of records—one for each file—that each contained the features and label of the corresponding file.

5.2 Learning Methods

In our experiments, we used several standard learning algorithms and ensemble learning methods. More specifically, we used the `scikit-learn` Python module [21] to implement the following:

- Naïve Bayes, assuming Gaussian feature distributions
- Logistic Regression using L_1 regularization
- Classification and Regression Tree (CART) with splits computed based on *entropy*
- Random Forest
- AdaBoost
- Bagging

Naïve Bayes is a popular classification algorithm derived from Baye's rule:

$$P(y|x_1, x_2, \dots, x_n) = \frac{P(y)P(x_1, x_2, \dots, x_n|y)}{P(x_1, x_2, \dots, x_n)}$$

where y is the label and $x_1, x_2, \dots, x_n$ are the values for the n features for a given record. When given a test record, a Naïve Bayes classifier predicts the label that, based on the feature values in the record, has the highest probability. This learning algorithm was chosen because Baye's rule parallels the detection rules in the study by Yonts [12]. The algorithm is considered "naïve" because each feature is assumed to be probabilistically dependent upon the label and independent of the other features. While not always true, this assumption allows for much more computationally feasible models and has still achieved high performance in a variety of other applications [22].

Ng argues that Logistic Regression will ultimately tend to have lower error rates than Naïve Bayes when enough data is used for training [23]. Whereas Naïve Bayes predicts $P(Label|Features)$ by estimating $P(Feature|Label)$ from the training data, Logistic Regression directly creates a function that models $P(Label|Features)$ from the training data. We used L_1 regularization in our implementation of Logistic Regression. This variant of the standard algorithm attempts to prevent overfitting, a situation in which a classification model erroneously reflects some of the noise in the training data in addition to the underlying relationship between the features and the label.

Decision Trees were the most effective classifier in a previous study on PE32 header based features [5]. This class of algorithm builds classifiers that make predictions by following a "tree" model, which can be thought of as a flow chart, based upon the records in the training data. We used the `scikit-learn` implementation of the CART decision algorithm, which builds trees by picking the feature and threshold with the highest *information gain* for each node in the tree. To measure information gain, we used *information cross-entropy*:

$$H(X_m) = \sum_k p_{mk} \log(p_{mk})$$

where the *cross-entropy* $H(X_m)$ of feature X on node m is a function of proportion p_{mk} of class k observations on node m [21].

In addition to these individual machine learning algorithms, we implemented three ensemble learning methods: random forests, AdaBoost, and Bagging. Ensemble learning methods have been used to mitigate performance loss due to class imbalance in other machine learning applications, with AdaBoosting and Bagging being particularly useful [15]. Random forests train a *forest* of multiple decision trees. The classifier makes predictions based on the average probabilistic predictions of each tree. We chose to implement this method due to its similarity to the decision trees that have yielded high performance in previous studies of PE32 header based features. AdaBoost trains a set of simple classifiers over several iterations of modified versions of the training data, and the resulting ensemble classifier makes predictions based on a weighted majority vote of the simple classifiers. Finally, Bagging iteratively trains a set of complex classifiers (as opposed to the simple ones used by AdaBoost) from random samples of the given training data [21]. Each of these learning methods was implemented with the default `scikit-learn` parameters.

5.3 The Machine Learning Framework

Each experiment was broken into a set of *trials* of four phases. First, we sampled the database for sets of training and test data. Then, we trained classifiers from each set of training data. Next, each classifier predicted the label for each record in a set of test data that was separate and independent of the training data the classifier was trained on. Finally, the performance of each classifier was assessed by comparing the predicted labels with the true labels of each record in the given set of test data.

Given a database, our sampling algorithm returns random training and test samples, which are collections of records from the database. The training and test samples do not have any records in common. We specify the desired number of records that each sample should contain[2], as well as the m, or *malware prevalence*, which is the percent, expressed as a decimal between 0 and 1, of records in the sample that correspond to malicious files. We separately specify m_{tr} and m_{te}, or the malware prevalence for the training and test samples, respectively. For each trial, we also provide a given number of random state *seeds*[3] (ten, unless specified otherwise), each of which produces one pair of training and test samples independently of the other seeds. Each seed is logged in case future sample reproduction is necessary. Each pair of samples is used in a sub-trial, in which a new, unique classifier is trained and tested with its given samples.

5.4 Performance Measurement

After being trained, classifiers predict the label for each record in the test sample by examining only the features contained in the record. These predictions are compared with the known true labels to calculate the *precision*, *recall*, and *F-measure* of the sub-trial. Each of these

[2] For this study, we always chose a 9:1 ratio of training records to test records. (31,500 and 3,500 total, unless specified otherwise.)

[3] Seeds specify the starting state of a random number generator and are useful for reproducing sequences of random numbers.

calculations provides an indication of the performance of the classifier in a sub-trial. Precision and recall are defined as follows:

$$precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

$$recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

where a "true positive" is a prediction for a record correctly predicted to be malicious, a "false positive" is one incorrectly predicted to be malicious, and a "false negative" is one incorrectly predicted to be benign. $TruePositives$, $FalsePositives$, and $FalseNegatives$ are the counts of true positives, false positives, and false negatives, respectively.

In malware detection, precision is the chance that a file predicted to be malware is actually malware, and recall is the chance that a malicious file is predicted to be malicious. While both of these measures can be useful in some situations, we primarily use them to calculate the generalized F-measure, the metric used to assess overall performance [24]:

$$F_\beta = \frac{(\beta^2 + 1) * precision * recall}{\beta^2 * precision * recall}$$

This metric reflects the harmonic mean, weighted by β, between precision and recall. F-measure ranges from 0 (worst) to 1 (best). We specify the value of β before the trial begins. Machine learning studies often use $\beta = 1$, corresponding to an equally weighted harmonic mean of precision and recall. In accordance with this standard practice, we used F_1 scores in our initial experiments.

In discussions with antivirus software engineers, we learned that commercial antivirus usually places a premium on minimizing false positives over maximizing true positives—vendors believe that most users would be harmed more by having their benign software quarantined or deleted than by having a piece of malware on their system. Some organizations and users may benefit by accepting a higher false positive rate in order to catch more malware, but this research is more focused on addressing the needs of the primary consumer based for standard antivirus software. Thus, in later experiments exploring class imbalance, we chose for β a value of 0.3333, which corresponds to an F-measure weighted 90% towards precision. For simplicity's sake, we denote this as F_β instead of $F_{0.3333}$. We noted, however, that all observed trends in the results were equally valid when using the neutral-weighted F_1 score.

We recorded the average number of true positives, false positives, true negatives, and false negatives over each sub-trial in a trial, as well as the parameters used for the trial: m_{tr}, m_{te}, number of records per sample, database, random seed, and machine learning algorithm.

6 Experiments

6.1 Learning Algorithm Comparison

6.1.1 Design.

This experiment tests the efficacy of PE32 header based features when using a variety of learning methods and malware prevalences. Each trial used of 22,500 training samples and 2,500 test samples. Across the trials, Naïve Bayes, Logistic Regression, and CART trees were compared over a variety of combinations of m_{tr} and m_{te}.

6.1.2 Results.

Figure 1 summarizes the results of the trials in this experiment. Each bar reflects the mean F_1 score performance of the sub-trials in the trial testing the specified combinations of m_{tr}, m_{te}, and learning algorithm. Trials employing decision trees always outperformed their Naïve Bayes and Logistic Regression counterparts at the same malware prevalences.

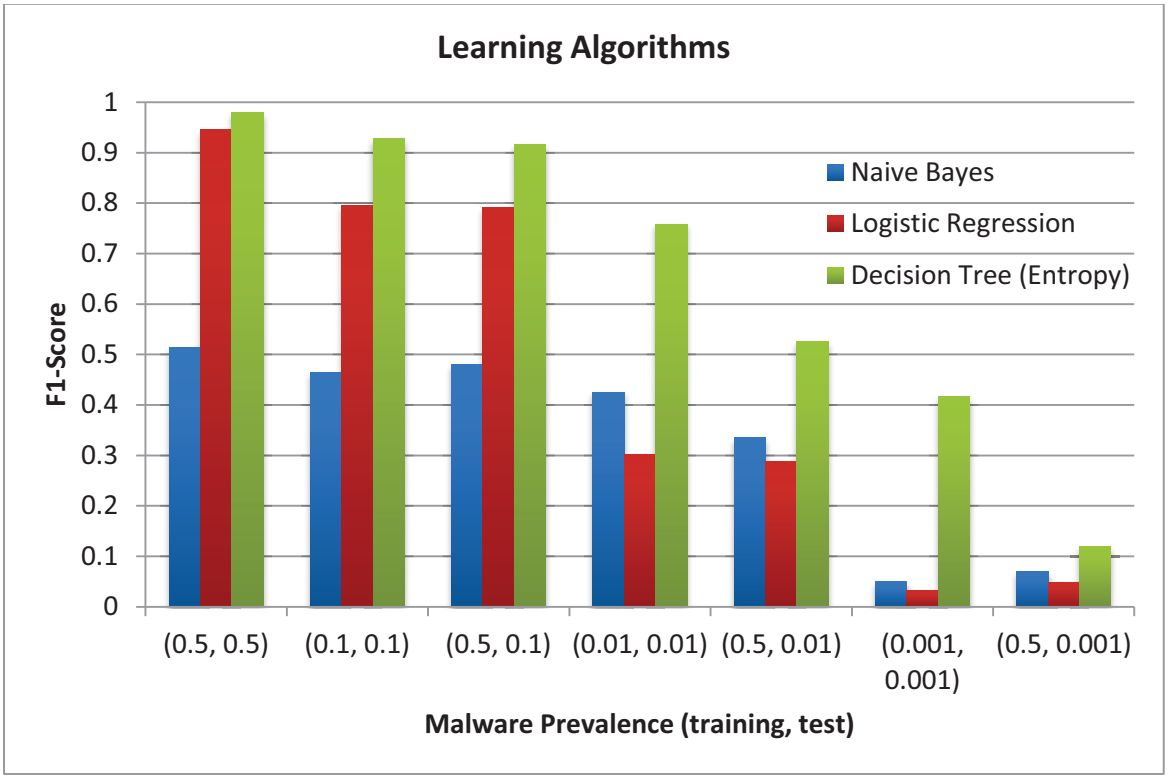

Figure 1 F_1 scores of trials with varying malware prevalence and learning algorithms. Tuples on the x-axis denote the m_{tr} and m_{te}, respectively.

6.1.3 Discussion.

When the training and test data were split evenly between malware and benign software (i.e. $m_{tr} = 0.5$ and $m_{te} = 0.5$), as is often the case of machine learning based malware detection research, our results for logistic regression and decision trees were competitive with any of the literature we reviewed. We cannot inarguably claim that our classifier performed "better" than those of other studies, because other studies used different databases and did not always use F-measures to assess performance.

Additionally, similar to how Yan et al. found that Decision Trees performed best for predicting the families of different malware [5], we find that classifiers trained by the CART algorithm performed best regardless of malware prevalence. Furthermore, Naïve Bayes outperformed Logistic regression as m_{te} decreased. This follows the finding of Ng and Jordan that Logistic Regression has a lower asymptotic error rate than Naïve Bayes, but Naïve Bayes generally performs better when less data is available (or, in this case, when less data about malware is available) [23].

6.2 Initial Malware Prevalence Investigation

6.2.1 Design.

Operational machine learning based malware detection programs need to be optimized for low m_{te}, as even computers with malware on them tend to contain more benign software than malware by several orders of magnitude. This is more difficult than $m_{te} = 0.5$, because the abundance of benign software tends to increase the false positive rate. Thus, we hypothesized that decreasing m_{te} would decrease F-measure performance. However, we initially hypothesized that performance would still be maximized by training with $m_{tr} = 0.5$, as it would provide as much information as possible about both malicious and benign software. This experiment compares the performance of trials across a variety of m_{tr} and m_{te} using primarily PE32 header based features to train decision trees—the highest performing algorithm in the previous experiment.

6.2.2 Results.

Figure 2 shows the results of the trials involved in this experiment. As m_{te} decreases, F_1 score performance decreases as well. Additionally, trials with $m_{tr} = m_{te}$ outperform their $m_{tr} = 0.5$ counterparts.

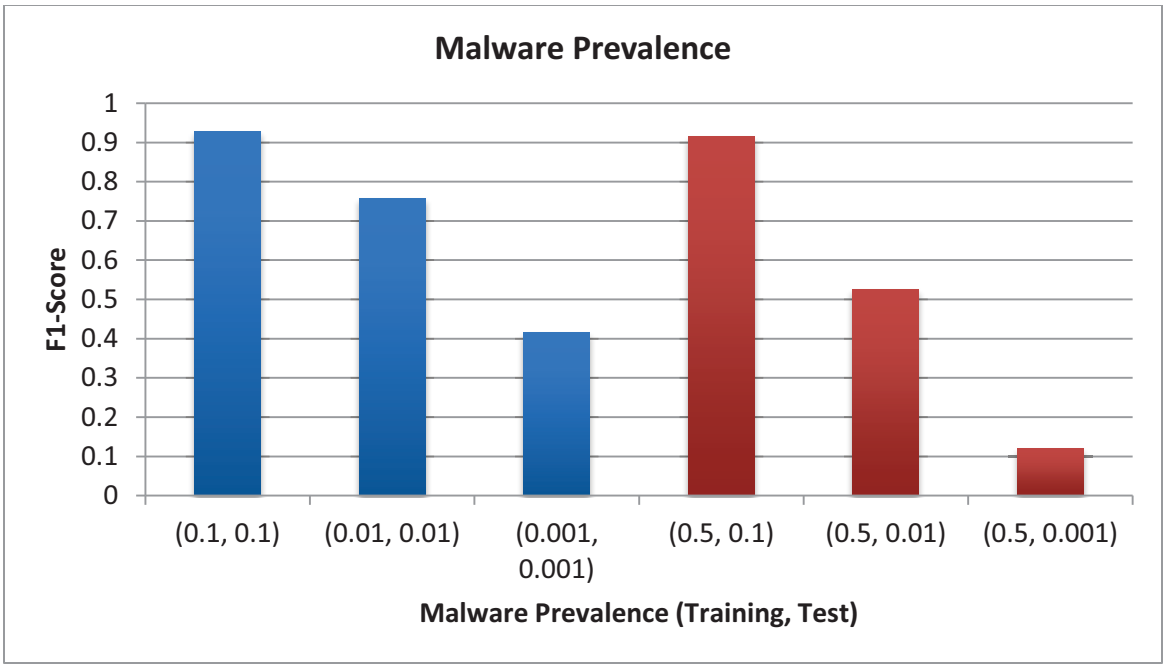

Figure 2 F_1 scores of trials over a variety of training and test malware prevalences. Blue bars correspond to trials where $m_{tr} = m_{te}$; red bars correspond to trials where m_{tr} is kept at 0.5. Tuples on the x-axis denote the m_{tr} and m_{te}, respectively.

6.2.3 Discussion.

These results support our first hypothesis: classifier performance drops precipitously with m_{te}. However, these trials also contradict our second hypothesis, because trials in which malware prevalence between the training and test data were the same outperformed trials with high m_{tr}. Furthermore, the differences in performance widened as m_{te} decreased. It could be that decision trees are optimized for testing on data with the same class imbalance as the training data, or it could be that there is some inherent quality to PE32 header based features (or malware and benign software in general) that causes a realistic m_{tr} to produce better classifiers than $m_{tr} = 0.5$.

6.3 Ensemble Learning and Class Imbalance

6.3.1 Design.

This experiment follows up on the results of the initial malware prevalence investigation. As mentioned previously, ensemble learning methods are a common and generally effective technique to assuage class imbalance in other applications [15], so they could be useful for either increasing overall performance for trials in which m_{te} is low or in reversing the trend that trials in which $m_{tr} = m_{te}$ outperform trials of the same m_{te} in which $m_{te} = 0.5$. However, we hypothesize that, when using features based on PE32 headers, ensemble learning methods do not significantly remedy the class imbalance problem in either of these ways. Accepting this hypothesis would suggest that more specifically tailored optimization might be required to

effectively detect malware in the wild. Alternatively, it might suggest that PE32 header data, while effective in malware classification at higher prevalences, should be replaced by some other feature class at lower malware prevalence.

We also compare the overall effectiveness of several ensemble methods to the effectiveness of CART decision tree classifiers, as a non-ensemble baseline. We chose CART decision tree classifiers because they performed best in our previous algorithm comparison experiment on PE32 header based features. Although other studies have used ensemble methods for malware classification [25], and they generally improve malware detection performance compared to traditional methods with roughly equal malicious and benign samples [5], ensemble methods have not been explicitly compared against single traditional classifiers in the context of reduced malware prevalence.

We ran a trial for every combination of sets of learning methods, m_{tr}, and m_{te}. Learning methods varied among CART, Random Forest, AdaBoost, and Bagging. In order to get a more robust idea about the relationship between m_{tr} and m_{te}, we varied them among 0.001, 0.01, 0.1, 0.25, 0.5, 0.75, and 0.9. We built a table for each algorithm displaying a trial's average F_β performance (where $\beta = 0.3333$) of any m_{tr} and m_{te}. The tables identify variations in performance as m_{tr} and m_{te} are varied, and show which m_{tr} yields optimal F_β for a given m_{te}.

6.3.2 Results.

Average classifier performance at each combination of learning method and malware prevalence is shown in Table 1. F_β scores tend to increase with m_{te}, and they tend to reach their peak within a given m_{te} when m_{tr} is roughly equal to it. While Random Forests and Bagging seem to score slightly higher, score differences between the various learning methods are too slight to be conclusive.

6.3.3 Discussion.

The results of this experiment show that, regardless of learning method or m_{tr}, classifiers perform poorly when m_{te} is realistically low, no matter how well they perform when $m_{te} = 0.5$. Furthermore, for a given learning method and m_{te}, classifiers usually perform best when m_{tr} is equal to, or nearly equal to, m_{te}. Thus, for optimal operational performance, a classifier should be trained on data with a realistically low proportion of malware.

Despite performing well regardless of class imbalance in other applications, ensemble learning methods fared no better at mitigating class imbalance than decision trees here. Furthermore, they did not perform significantly better overall than decision trees. Thus, our hypothesis is confirmed, and ensemble methods do not alleviate our concerns about class imbalance for malware detection.

F_β results from training on PE Header Features

Adaboost								Decision Trees							
	m_{tr}								m_{tr}							
m_{te}	0.001	0.01	0.1	0.25	0.5	0.75	0.9	0.001	0.01	0.1	0.25	0.5	0.75	0.9	m_{te}
0.001	0.470	0.399	0.187	0.090	0.051	0.021	0.012	0.541	0.403	0.176	0.106	0.068	0.038	0.021	0.001
0.01	0.721	0.823	0.670	0.455	0.315	0.168	0.099	0.827	0.813	0.649	0.503	0.383	0.263	0.159	0.01
0.1	0.800	0.946	0.946	0.896	0.832	0.688	0.546	0.873	0.952	0.946	0.914	0.871	0.793	0.673	0.1
0.25	0.804	0.955	0.975	0.959	0.935	0.867	0.786	0.877	0.964	0.976	0.967	0.951	0.919	0.862	0.25
0.5	0.806	0.958	0.984	0.983	0.977	0.951	0.918	0.877	0.968	0.987	0.986	0.982	0.971	0.950	0.5
0.75	0.806	0.959	0.988	0.991	0.990	0.983	0.970	0.878	0.969	0.990	0.993	0.993	0.989	0.982	0.75
0.9	0.806	0.959	0.988	0.993	0.994	0.993	0.990	0.878	0.969	0.991	0.995	0.996	0.996	0.993	0.9

Bagging								Random Forest							
	m_{tr}								m_{tr}							
m_{te}	0.001	0.01	0.1	0.25	0.5	0.75	0.9	0.001	0.01	0.1	0.25	0.5	0.75	0.9	m_{te}
0.001	0.539	0.472	0.240	0.143	0.094	0.049	0.025	0.458	0.584	0.262	0.155	0.098	0.052	0.025	0.001
0.01	0.729	0.859	0.731	0.590	0.473	0.312	0.183	0.584	0.883	0.760	0.612	0.484	0.329	0.182	0.01
0.1	0.815	0.956	0.961	0.938	0.906	0.833	0.709	0.643	0.957	0.965	0.943	0.910	0.842	0.709	0.1
0.25	0.820	0.965	0.982	0.975	0.964	0.936	0.881	0.644	0.963	0.983	0.978	0.966	0.939	0.882	0.25
0.5	0.820	0.967	0.989	0.989	0.987	0.977	0.958	0.650	0.965	0.989	0.990	0.987	0.979	0.957	0.5
0.75	0.821	0.969	0.991	0.994	0.994	0.992	0.985	0.648	0.966	0.992	0.995	0.995	0.992	0.985	0.75
0.9	0.821	0.969	0.992	0.996	0.997	0.997	0.995	0.649	0.966	0.993	0.996	0.997	0.997	0.995	0.9

Table 1 F_β results from training on primarily PE32 header features. Within each row for each algorithm, blue indicates relatively good performance, and red indicates relatively poor performance.

6.4 Windows API Import Features

6.4.1 Design.

Windows API imports are statistically different between malware and benign software [9], and so could potentially make viable features for machine learning classification. Repeating the previous experiment's ensemble learning trials on a database of features based on Windows API imports, the next experiment examines whether or not the class imbalance trends are unique to PE32 header based features. We predict that class imbalance is a problem for malware detection generally, and thus hypothesize that similar class imbalance trends would be observed for Windows API import features as for header features.

In addition to examining class imbalance trends, we intend to compare the performance of classifiers built on API import features to the performance of classifiers built on PE32 header based features.

This experiment used the 2,067 features describing whether each Windows API was or was not imported statically in a given PE32 file. The classifiers were trained on these features as is, without transforming them or weeding out any unnecessary ones. While such techniques might

optimize overall performance, they are not likely to mitigate the effects of class imbalance and thus would not contribute to this experiment.

Because these features are run through the same trials as the previous ensemble learning experiment, the results can be directly compared to those for PE32 header features at any given m_{tr} or m_{te}.

6.4.2 Results.

Table 2 shows the average classifier performance from training on Windows API import features. As with PE32 header features, F_β scores tended to increase with m_{te}, and, within one m_{te}, optimal scores usually occurred when m_{tr} nearly equalled m_{te}. Adaboost was the biggest exception: for five different values of m_{te}, $m_{tr} = 0.1$ produced the optimal score.

Furthermore, the best classifier performance of each m_{te} and learning method using PE32 header data is shown in Figure 3. Similarly, the best classifier performance of each m_{te} and learning method using Windows API imports is shown in Figure 4. Although the classifiers performed fairly well at higher malware prevalence, they also performed poorly when m_{te} was low, regardless of which feature type they were trained from. Furthermore, the PE32 header data yielded better performance than static Windows API imports at the same m_{te} in every case. Performance trends did not vary greatly across the four machine learning algorithms applied.

F_β results from training on WinAPI Features

Adaboost								Decision Trees							
	m_{tr}								m_{tr}							
m_{te}	0.001	0.01	0.1	0.25	0.5	0.75	0.9	0.001	0.01	0.1	0.25	0.5	0.75	0.9	m_{te}
0.001	0.077	0.066	0.096	0.03	0.02	0.002	0.002	0.168	0.168	0.1	0.062	0.046	0.002	0.002	0.001
0.01	0.369	0.384	0.45	0.227	0.136	0.018	0.016	0.333	0.604	0.483	0.368	0.277	0.019	0.018	0.01
0.1	0.35	0.544	0.841	0.725	0.619	0.163	0.151	0.458	0.859	0.88	0.839	0.79	0.178	0.168	0.1
0.25	0.335	0.565	0.892	0.849	0.805	0.369	0.348	0.461	0.883	0.929	0.916	0.898	0.394	0.376	0.25
0.5	0.338	0.585	0.909	0.906	0.896	0.635	0.614	0.479	0.894	0.945	0.946	0.941	0.659	0.642	0.5
0.75	0.333	0.592	0.916	0.925	0.933	0.838	0.826	0.479	0.898	0.951	0.956	0.957	0.853	0.842	0.75
0.9	0.331	0.593	0.918	0.932	0.944	0.938	0.934	0.478	0.9	0.953	0.96	0.963	0.945	0.941	0.9

Bagging								Random Forest Classifiers							
	m_{tr}								m_{tr}							
m_{te}	0.001	0.01	0.1	0.25	0.5	0.75	0.9	0.001	0.01	0.1	0.25	0.5	0.75	0.9	m_{te}
0.001	0.154	0.228	0.136	0.079	0.063	0.002	0.002	0.231	0.254	0.161	0.102	0.068	0.002	0.002	0.001
0.01	0.324	0.669	0.55	0.44	0.354	0.02	0.019	0.253	0.656	0.6	0.487	0.372	0.02	0.018	0.01
0.1	0.384	0.865	0.897	0.87	0.838	0.181	0.171	0.291	0.843	0.908	0.884	0.845	0.181	0.17	0.1
0.25	0.369	0.881	0.937	0.927	0.918	0.4	0.382	0.289	0.862	0.94	0.932	0.921	0.399	0.381	0.25
0.5	0.377	0.889	0.948	0.95	0.949	0.664	0.649	0.298	0.868	0.948	0.952	0.949	0.664	0.648	0.5
0.75	0.37	0.892	0.952	0.958	0.96	0.856	0.847	0.299	0.871	0.951	0.958	0.96	0.856	0.846	0.75
0.9	0.367	0.893	0.954	0.961	0.964	0.946	0.943	0.293	0.872	0.952	0.961	0.964	0.946	0.942	0.9

Table 2 F_β results from training on Windows API import features. Within each row for each algorithm, blue indicates relatively good performance, and red indicates relatively poor perforamce.

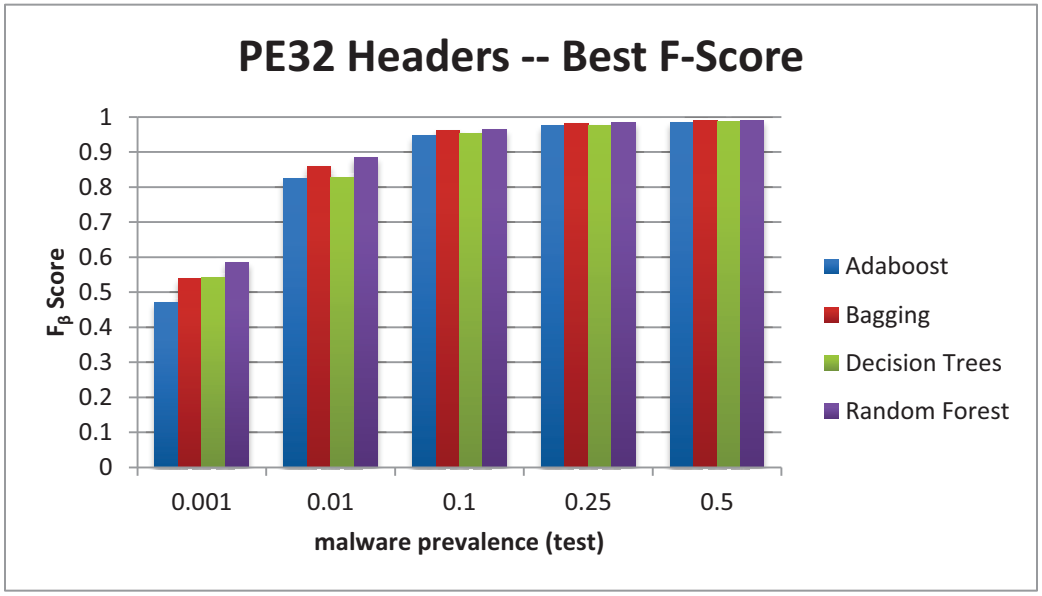

Figure 3 Performance of PE32 header data at varying *test* malware prevalence. In each column, the F_β score is the maximum achieved at any *training* malware prevalence.

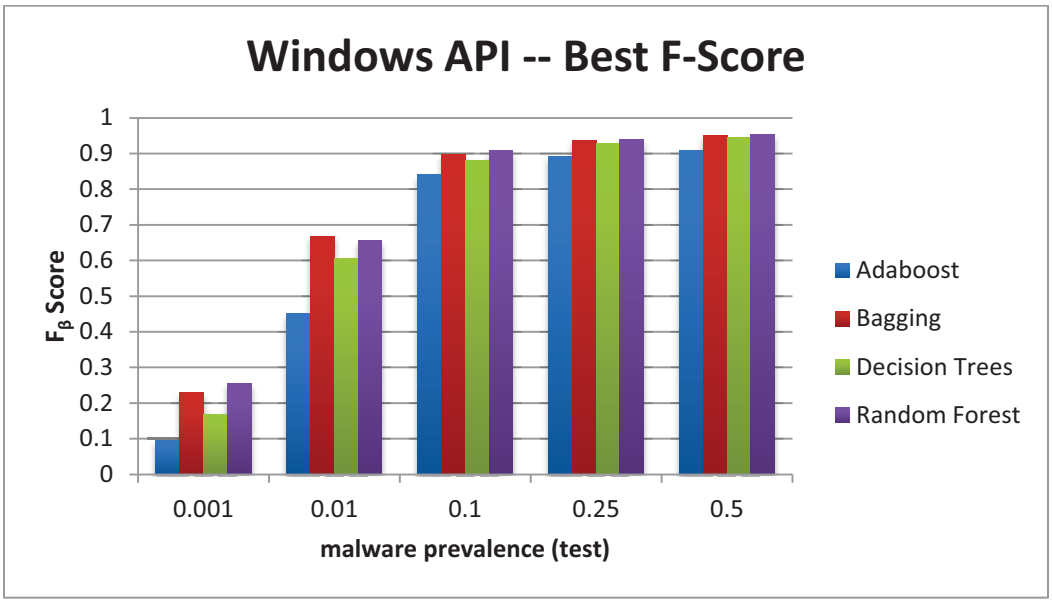

Figure 4 Performance of Windows API imports at varying *test* malware prevalence. In each column, the F_β score is the maximum achieved at any *training* malware prevalence.

6.4.3 Discussion.

While less strictly so, classifiers trained on import features still achieve optimal performance when m_{tr} is about equal to m_{te}. AdaBoost was the largest exception, in which the optimal performance is at $m_{tr} = 0.1$ for each m_{te} in the set $\{0.001, 0.01, 0.1, 0.25, 0.5\}$. Yet even this exception does not significantly deviate from the trend, as 0.1 is the median of that set of test malware prevalences. Thus, the primary hypothesis behind this experiment is supported by its results.

Additionally, these results show that PE32 header features outperformed Windows API import features at any given learning method or malware prevalence: at each m_{te}, even the worst performing learning method trained by header features (Figure 3) scored higher than the best performing learning method trained by API import features (Figure 4). While neither of these features were optimized by any feature transformations, this comparison is still useful as a novel baseline comparison between these two feature categories.

It should be noted that Windows API imports extracted through static analysis are not expected to be useful in operational classifiers for the long term—malware authors could use suspicious operating system functions through dynamic methods that thwart static analysis [10], which would make malware look nearly identical to benign software under these features. The import features for our malware are still significantly different than those of our benign software, but this trend could change as malware authors adapt. This potential weakness does not affect the goals of this experiment, but it is worth bearing in mind as a disadvantage to relying upon import features in the future.

6.5 The Effect of m_{tr} on Feature Utility

6.5.1 Design.

Other studies which compare the utility of various features for malware detection do not consider the proportion of malware reflected in their training and test samples; it is implicitly assumed that the usefulness of features for machine learning based malware detection are independent of the m_{tr} used to train the classifier. This experiment is designed to test this assumption; we hypothesize that the relative value of a PE32 header feature for malware detection depends upon the proportion of the training data that is malware. In other words, we hypothesize that features that are useful at higher malware prevalence can be different from the features that are useful at lower malware prevalence. If this hypothesis is verified, it follows that a classifier should not be trained using features that were only found to be useful at a higher m_{tr} if the classifier is intended to detect malware when trained at a low m_{tr}.

The importance of this hypothesis hinges on the finding that classifiers perform best when the training data has the same proportion of malware as the test data—a finding we have already verified for both PE32 header based features and API import based features. Because benign software greatly outnumbers malware in practical settings, an optimal operational classifier should train at a low m_{tr}.

In these trials, we separated our database of primarily header based features into several databases of isolated features, in which records for each contained just a file's label, and the value of the one isolated feature in the database. We ran trials on each database using 30 sub-trials (instead of the typical 10) and the random forest classifier—the classifier which arguably performed the best, however slightly, among the methods used in the ensemble learning experiments. One trial was performed for every combination of m_{tr} and m_{te} from the set $\{0.001, 0.01, 0.1, 0.25, 0.5\}$. We examined the classifier performances at $m_{te} = 0.01$, which is presumably low enough to be comparable to a realistic setting.

These results were used to perform a *two-way ANOVA with replication*. A two-way ANOVA is a statistical test that allows us to isolate the variance in the F_β performance of two *factors*, or independent variables. The factors used in this experiment are "feature used" and "m_{tr}". Each factor can take on one of several values, or *categories*. For our test, the different categories of the Feature factor were the 40 different features based primarily on PE32 header information, and the categories of the m_{tr} factor were 0.001, 0.01, 0.1, 0.25, and 0.5.

Furthermore, by using *replication*, which entails providing the ANOVA calculation with the results of each of the 30 sub-trials of each trial, we can evaluate the *interaction* of the two features on performance. If an interaction exists, then the variations in performance across the different categories within a factor depend upon the category of the other factor.

The primary result of an ANOVA test is the *p-value* calculated for each *source of variation*. Sources of variation include both factors and the interaction. The p-value for factors reflects the probability that performance variations between factor categories are merely the result of random chance; for interaction, the p-value reflects that probability that variations observed between the

two factors' categories are independent of each other. Any p-value below 0.05, which indicates a probability below 5%, is significant enough to rule out the possibility that the variations in question are due to random chance only.

For this experiment, we are primarily interested in the p-value associated with the interaction of Feature and m_{tr}. If less than 0.05, this p-value implies that the variations in performance between different features depends upon m_{tr}. In other words, a significant p-value for interaction means that the relative utility of a feature cannot be considered without also considering the malware prevalence in the classifiers' training data.

6.5.2 Results.

Table 3 summarizes the results of the ANOVA test. The column *Source of Variation* contains the name of the source of variation assessed in each row.[4] The *SS* column contains the sum of squares of each source of variation. The *df* column reports the degrees of freedom, or the number of values in the calculation that are free to vary, for each source of variation. The *MS* column contains the mean squares, or estimated population variance, of each source of variation; MS is calculated by dividing SS by df. The *F* column refers to the "F-statistic", which corresponds to how much variance occurs between different trials versus how much variance occurs within the sub-trials of each trial; the F-statistic forms the basis for rejecting the possibility that the variations of a source of variation are due to random chance only. The p-value (shown in column of the same name) will be below 0.05—the value used to indicate significance—when the F-statistic is above the *critical F-statistic*, which is shown in the *F crit* column.

For this experiment, only the interaction p-value needs consideration. Because the p-value is 0^5, we confidently conclude that the differences in performance between the various PE32 header factors cannot be considered without also considering the m_{tr} the classifier was trained at.

Table 4 helps to illustrate the results of the ANOVA test; it lists the 10 features [6] which, when used in isolation for training at the given m_{tr}, produced the top performing classifiers on average when tested on a $m_{te} = 0.01$ sample. A qualitative examination of these lists supports the ANOVA results: these two lists only share four features in common, and of those common features, one, *NumLinenums!=0*, actually performs better when $m_{tr} = 0.5$ versus when $m_{tr} = 0.01$.

---

[4] "Within" refers to the *within-group* factors, which examines the variation within the 30 sub-trials of each trial. "Total" examines the variation among all observations. Neither are pertinent for this experiment; they are only retained for completeness.

[5] The p-value might not literally equal 0, but the precision of Microsoft Excel 2010 cannot distinguish it from 0.

[6] Most of these features reflect the literal header value for which they are named, which are explained in the PE specification [11]. Exceptions include those that end in *!=0*, which are features that equal 1 if any of the sections in the PE32 file have a nonzero value for that header data. For instance, *NumLinenums!=0* is true for a file if that file has any sections in which the NumberOfLineNumbers field is nonzero. Additionally, samplesize is the actual size of the PE32 file, and HighEntropy is a feature set to 1 when any section of a file has an entropy greater than 7.

2-way ANOVA with replication of feature and m_{tr}

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
m_{tr}	7.268943	4	1.817236	309.0962	2.2E-241	2.373464
Feature	53.10001	39	1.361539	231.5861	0	1.401407
Interaction	14.77306	156	0.094699	16.1075	0	1.196534
Within	34.09931	5800	0.005879			
Total	109.2413	5999				

Table 3 Results of a 2-way ANOVA with replication of feature and m_{tr}. Performance is measured by F_β at $m_{te} = 0.01$. The highlighted cell is the p-value of the interation source of variation.

Top performing isolated PE header features

$m_{tr} = 0.01$			$m_{tr} = 0.5$		
Feature	Mean $F_{0.33}$	Std dev	Feature	Mean $F_{0.33}$	Std dev
BaseOfCode	0.6039	0.1411	BaseOfCode	0.5249	0.1105
PointerToLinenumbers0	0.2826	0.1698	BYTES_REVERSED_HI	0.3296	0.0974
NumLinenums0	0.2357	0.1843	BYTES_REVERSED_LO	0.3282	0.0974
PointerToRelocations0	0.2012	0.1515	NumLinenums0	0.2569	0.1505
BaseOfData	0.1789	0.1354	PointerToLinenumbers0	0.2433	0.1117
PointerToSymbolTable	0.1659	0.1613	RELOCS_STRIPPED	0.237	0.024
AddressOfEntryPoint	0.1608	0.0795	PointerToRelocations0	0.1816	0.1401
NumberOfSymbols	0.1293	0.1513	SizeOfStackCommit	0.1518	0.0504
samplesize	0.1035	0.0741	HighEntropy	0.1472	0.0215
SizeOfInitializedData	0.0996	0.1336	LOCAL_SYMS_STRIPPED	0.147	0.0132

Table 4 The highest performing isolated features, tested at $m_{te} = 0.01$. Mean F_β performance is shown, as well as the standard deviation of the F_β performances among the sub-trials of each trial. Highlighted cells indicate features that are on both lists.

6.5.3 Discussion.

This experiment verifies our hypothesis that the relative utility of our primarily header based features depends upon the malware prevalence of the training sample. Previous machine learning based malware detection research has generally assumed that, if a feature is found to successfully train a classifier at a high m_{tr}, then the feature will remain relatively informative regardless of m_{tr}. However, the results of this experiment contradict this assumption. Thus, the only way to demonstrate that a feature is useful when malware is a small proportion of the training data is to test it in that situation.

7 Conclusion

This research contributes to the development of machine learning based malware detection through several major stages. First, it explores the usefulness of features based on PE32 header data, finding that, if ensemble learning methods or decision trees are used for learning, they compete well with other features that have been examined in the literature. Other studies have noted the potential usefulness of PE32 header data before, but our research is the first to use this feature source to build a machine learning classifier that discriminates between malware and benign software in general.

However, while features based primarily on PE32 header data led to high performance when there was no class imbalance in the training and test data—the norm for academic literature in the field—these features performed abysmally in trials where test data had a realistically low proportion of malware. This led us to explore the issue of class imbalance as it applies to malware detection, an issue that we have not seen the literature address. Through our experiments, we found that classifiers tend to achieve optimal performance when the prevalence of malware in the training data roughly equals the prevalence of malware in the test data. This trend held regardless of learning method used and for two distinct sets of features: features derived from PE32 header data and features derived from static Windows API function imports. Even when using ensemble learning methods, which have been able to classify relatively well regardless of class imbalance in other machine learning applications, they did not appreciably mitigate this trend. Thus, if machine learning based malware detection software is to be used for operational malware detection, the classifiers in that software should train on data with a realistically low proportion of malware. That said, even our best classifiers did not perform satisfactorily on test data with a low malware prevalence, so further research is necessary before machine learning based malware detection can substantially augment current antivirus software.

The final contribution of this research lies in its investigation of the relationship between training malware prevalence and the usefulness of features for malware detection. Previous work has implicitly assumed that the usefulness of a feature is independent of the malware prevalence of training data. However, the experiments in this research show that training malware prevalence significantly impacts the relative utility of features derived primarily from PE32 header data, contradicting the aforementioned assumption. Thus, in order to demonstrate that a feature is useful for malware detection when using a particular malware prevalence for training, it must be demonstrated as such at that particular malware prevalence.

This finding will be important as long as optimal malware detection classifiers can only be obtained by training at the same malware prevalence that one expects to test at. Since malware is relatively rare in practical settings, classifiers must be trained on data in which malware is equally rare. To show that a feature is useful for practical malware detection, then, its relative utility must be demonstrated on training and test data with a realistically low malware prevalence. Furthermore, to show that one class of features is better than another for training machine learning based malware detection classifiers, the advantage of a class must be demonstrated over a range of training and test malware prevalences. Our comparison between

features derived from header data and API function imports, in which features derived from header data performed substantially better regardless of malware prevalence, provide an example of such a comparison. Without following this procedure, further studies cannot convincingly highlight a set of features for their practical effectiveness in machine learning based malware detection.

8 References

[1] Kotadia, “Eighty Percent of New Malware Defeats Antivirus.” Jul-2006.

[2] iMPERVA, “Assessing the Effectiveness of Antivirus Solutions,” *iMPERVA Hacker Intell. Initiat. Mon. Trend Rep.*, vol. 2012, no. 14, 2012.

[3] E. Nasi, “Bypass Antivirus Dynamic Analysis: Limitations of the AV model and how to exploit them,” Aug. 2014.

[4] K. Rieck, T. Holz, and C. Willems, “Learning and classification of malware behavior,” *Proc. 5th Int. Conf. Detect. Intrusions Malware, Vulnerability Assess. (DIMVA ’08)*, 2008.

[5] G. Yan, N. Brown, and D. Kong, “Exploring Discriminatory Features for Automated Malware Classification,” pp. 41–61, 2013.

[6] J. Z. Kolter and M. a Maloof, “Learning to Detect and Classify Malicious Executables in the Wild,” *J. Mach. Learn. Res.*, vol. 7, pp. 2721–2744, 2006.

[7] I. Santos, Y. Penya, J. Devesa, and P. Bringas, “N-grams-based File Signatures for Malware Detection.,” *ICEIS (2)*, pp. 317–320, 2009.

[8] T. Stibor, “A Study of Detecting Computer Viruses in Real-Infected Files in the n -gram Representation with Machine Learning Methods,” in *23rd International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2010, Cordoba, Spain, June 1-4, 2010, Proceedings, Part I*, 2010, pp. 509–519.

[9] M. Belaoued and S. Mazouzi, “Statistical Study of Imported APIs by PE Type Malware,” in *Advanced Networking Distributed Systems and Applications (INDS), 2014 International Conference on*, 2014, pp. 82–86.

[10] B. Blunden, *Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones & Bartlett Learning, 2013.

[11] Microsoft, “PE and COFF Specification.” 2013.

[12] J. Yonts, “Attributes of Malicious Files,” *SANS Inst. InfoSec Read. Room*, 2012.

[13] A. Walenstein, D. J. Hefner, and J. Wichers, “Header information in malware families and impact on automated classifiers,” *2010 5th Int. Conf. Malicious Unwanted Softw.*, pp. 15–22, Oct. 2010.

[14] G. Wicherski, “pehash: A novel approach to fast malware clustering,” *2nd USENIX Work. Large-Scale Exploit. anEmergent Threat. Botnets, Spyware, Worms, More*, 2009.

[15] X. Guo, Y. Yin, C. Dong, G. Yang, and G. Zhou, “On the Class Imbalance Problem,” *2008 Fourth Int. Conf. Nat. Comput.*, pp. 192–201, 2008.

[16] Various, “Open Malware Repository.” 2014.

[17] “LibreOffice Suite 4.3.1.” 2014.

[18] “FileZilla: The free FTP Solution.” 2014.

[19] E. Carrera, “\texttt{pefile} Python module.” 2014.

[20] Microsoft, “Alphabetical list of Win32 and COM API.” .

[21] F. Pedregosa, et al., “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.

[22] V. M. Telecommunications and V. Metsis, “Spam Filtering with Naive Bayes -- Which Naive Bayes?,” in *Third Conference on Email and Anti-Spam (CEAS*, 2006.

[23] A. Ng and M. Jordan, “On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes,” *Adv. neural Inf. Process. ...*, 2002.

[24] M. Sokolova, N. Japkowicz, and S. Szpakowicz, “Beyond accuracy, F-score and ROC: a family of discriminant measures for performance evaluation,” *AI 2006 Adv. Artif. ...*, 2006.

[25] P. Vinod, V. Laxmi, M. S. Gaur, S. Naval, and P. Faruki, “MCF: Multicomponent features for malware analysis,” *Proc. - 27th Int. Conf. Adv. Inf. Netw. Appl. Work. WAINA 2013*, pp. 1076–1081, 2013.